後編

組み込みシステム設計者のための LIN 2.0マイコン実装術

――スレーブ・ノード用ドライバ・ソフトウェア開発のポイント

システムの記事

舘 伸幸

前編（本誌2005年11月号，pp.92-97）と中編（同12月号，pp.96-105）でLIN 2.0のプロトコル仕様について説明しました．今回は，このLIN 2.0プロトコルを実際にマイコンに実装するためのドライバ・ソフトウェアの設計について，事例を交えて解説します．（筆者）

前編で解説したように，LINは大きく分けて「マスタ・タスク」と「スレーブ・タスク」という二つの機能で動作します．マスタ・タスクは基本的にスケジュールに従ってヘッダを送信するのみであるのに対し，スレーブ・タスクはヘッダを受信し，その内容に従ってレスポンスを送信または受信しなければなりません（**図1**）．さらに，スレーブ・タスクが単体で搭載されるスレーブ・ノードは，コストを低く抑えるために非力なマイコンで実現することがほとんどです．また，*RC*発振器など，精度の低いクロックによる動作を補正する必要が生じる場合もあります．したがって，ドライバ・ソフトウェアの設計にあたっては，マイコンの機能をうまく使いこなすくふうを施さなければなりません．

一方，マスタ・タスクは，LINプロトコルの実装よりは，むしろゲートウェイとしての動作や，OS（またはスケジューラ）との連携といったところに設計のポイントがあります．実際の動作環境ではLINクラスタが単独で動作することは少なく，マスタ・ノードは別の基幹回線（例えば，CANなど）とのゲートウェイとしての動作が主であるためです（**図2**）．

本稿では，LINプロトコルを実装するうえで，マイコンのハードウェア制御に関する技術的ポイントが多く，また実際にノードを設計する機会がより多いスレーブ・ノード（タスク）について解説します．

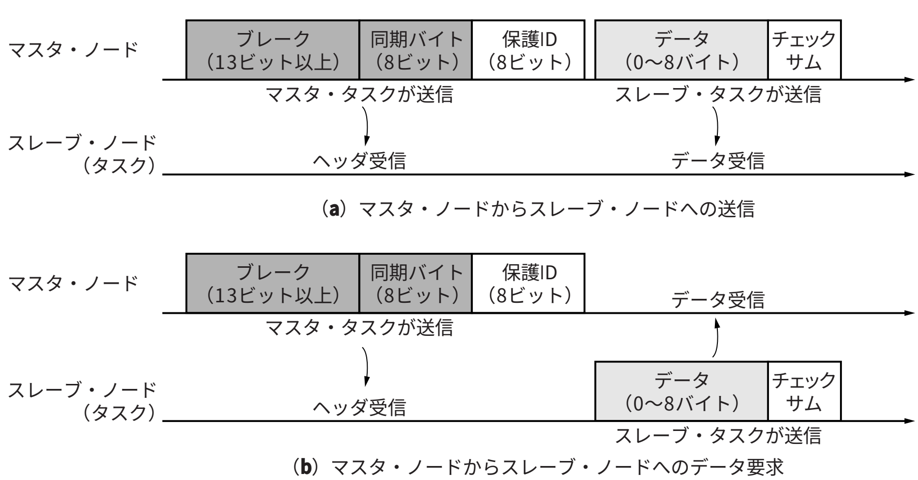


図1　マスタ・スレーブ間通信

マスタ・タスクがヘッダを送信して，スレーブ・タスクがそれを受信することで，後に続くデータの送受信を行う．

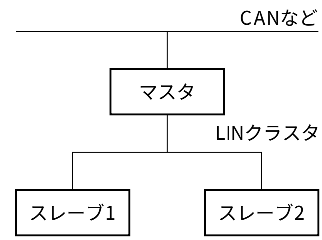


図2　実際の使用環境の例

とくに車載用途では，LINクラスタはCANのノードとして動作することがほとんどである．

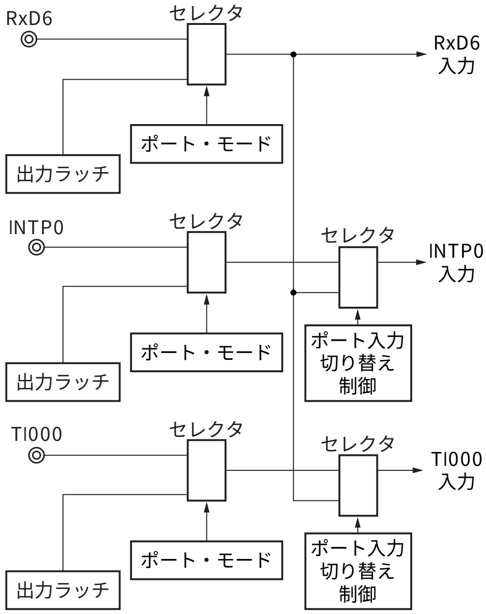


図3　LIN受信操作用ポートの構成（概略図）
ポート入力切り替え制御によって，マイコンの端子を結線することなく，受信ポート（RxD6）の入力データを外部割り込み（INTP0）や16ビット・タイマ/イベント・カウンタ（TI000）へ接続できる．

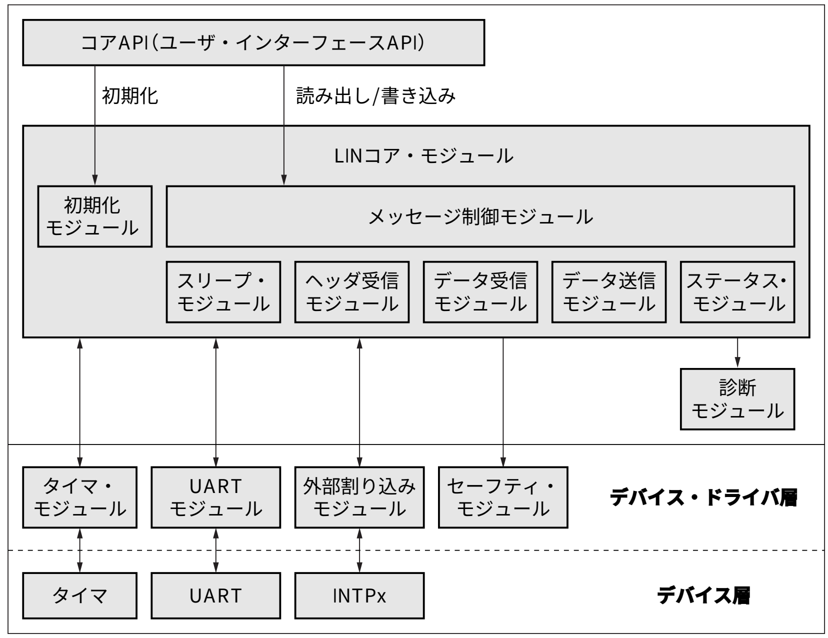


図4　プログラムの構造
移植性，拡張性などを考慮して，階層化と部品化を行っている．

1．ノードの構成と状態遷移

今回，ノードの開発に使用したのは，NECエレクトロニクスの8ビット・マイコン「μPD78F8006」（78K0/Kx1, Kx2シリーズ）です．UARTにLINのブレーク送受信を行う回路が含まれています（**図3**）．また，LIN仕様に適合した回線ドライバ回路も内蔵しているため，**写真1**のように小さなノード基板を作ることができます．

本稿では，ブレーク受信について，専用回路を使わない方法も記載します．ほかのマイコンを利用する場合は機能名などを随時読み替えて参照してください．

● 機能ごとにモジュール化する

図4に，LINドライバ（ソフトウェア）の構成例を示します．最下層はマイコンのハードウェアです．タイマ，UART，外部割り込み（INTPx）を使います．用途はそれぞれ，次のとおりです．

- タイマ　：各種時間規定の実現
- UART　：ヘッダの一部の受信とレスポンスの送受信
- INTPx　：ブレークの検出，同期バイトの計測，ウェイクアップの検出

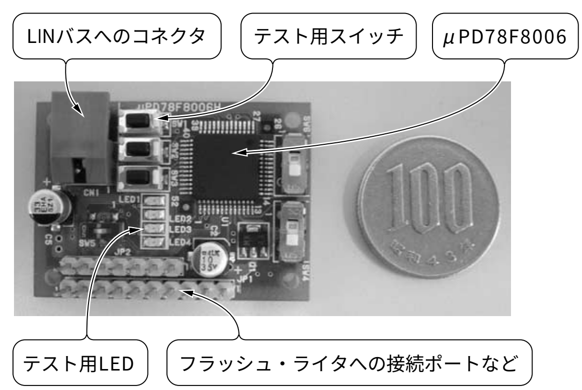


写真1　LINスレーブ・ノード基板の例

下から2層目には物理リソースのドライバを配置しています．この層でマイコンごとのハードウェア仕様の差異を吸収することにより，移植性を確保しています．

LINプロトコルそのものはLINコア・モジュールという部品で実現しています．このモジュールでは，LINスレーブ・タスクの基本機能であるヘッダ受信，データ（レスポンス）送受信，スリープ制御などを行います．メッセージ制御モジュールが，これらのモジュールを制御して実際の通信を行います．

そのほか，初期化および全体の状態管理を行うステータス・モジュールを備えています．診断モジュールは，クラ

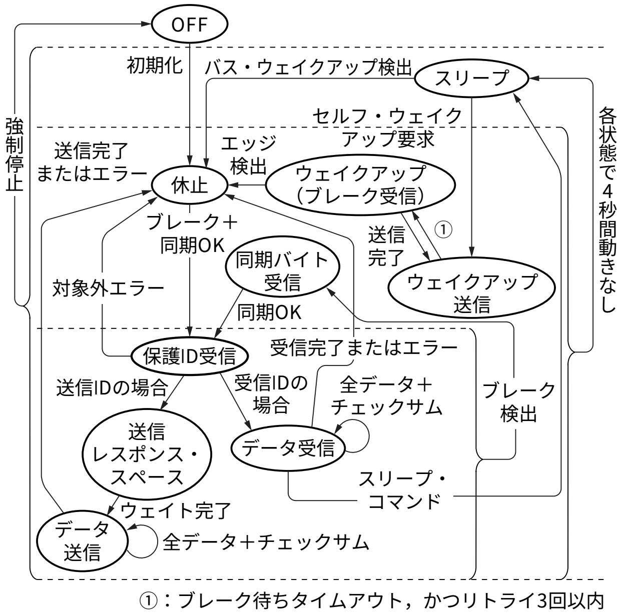


図5　LINコア・モジュールの状態遷移

基本的に，LIN Specification Package Revision 2.0の状態遷移仕様と対応しているが，スリープに関する状態遷移を追加している点が異なる．

スタごとの診断仕様に合わせてカスタマイズする必要があるため，LINコア・モジュールからは独立した構造にしています．

図5に，LINコア・モジュールの状態遷移図を示します．これは，基本的にLIN Specification Package Revision 2.0の状態遷移仕様（本稿の中編を参照）と対応していますが，大きな相違点としてはスリープに関する状態遷移を追加していることが挙げられます．

以下，主要な遷移条件に沿って設計のポイントを述べます．

2．LIN最大の特徴――ヘッダの処理

OFF状態から初期化を経て休止（dominant）状態になると，そこでマスタからのヘッダを待ちます．休止状態から保護ID受信状態へは，ブレークと同期バイトの受信で遷移します．ここが，マイコンでLINを実現するための大きな技術的ポイントです．

ヘッダの構成を**図6**に示します．ブレークはフレーム開始を示す13ビット以上のロー・パルス（“L”レベル）からなります．また，同期バイトは0x55に相当する01010101の

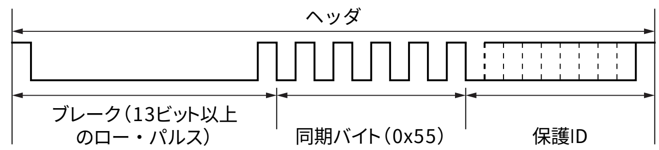


図6　ヘッダの構成

ヘッダは，マスタ・タスクによって送信されるフレームの最初の部分で，ブレーク，同期バイト，保護IDからなる．ブレークはフレーム開始を示す13ビット以上のロー・パルス（“L”レベル）．同期バイトは0x55に相当する01010101のビット列．

ビット列です．

● ブレーク検出の三つの方法

まず，ブレークをどのように検出するかですが，これには三つの方法が考えられます．

一つ目は，UARTのフレーミング・エラーを利用する方法です[注1]．この場合の注意点は，ノードのマイコンはかならず水晶などの精度の高い発振器で動作していなければならないことです．

実装例を**リスト1**に示します．これは，LIN 2.0仕様にある，フレーム途中のブレーク検出にフレーミング・エラーを利用している場合のものです．この関数はエラー発生時の割り込みハンドラです[注2]．ブレークの判定は16～17行目（薄い灰色の部分）で行っています．UARTがフレーミング・エラーで，かつ受信データがゼロであれば，ブレーク検出を意味します．ただし，フレーム途中のロー・パルス検出は，かならずしもブレークとはいえない場合があるので，対策が必要です．イベント・トリガ・フレーム時のレスポンスの衝突で，ロー・パルスが合成されてしまうことがあるためです．詳細は次の同期バイト処理のところで述べます．

フレーム途中のブレークはエラー発生とみなす必要がありますが，イベント・トリガ・フレームにおける衝突は，スレーブとしては単に無視するだけです．19行目（濃い灰色で示した部分）はその判断を行っており，その後，同期バイトの処理を呼んでいます．

二つ目のブレーク検出の方法は，ハードウェアによるブレーク検出機能を利用することです．今回使用したマイコンに搭載されているLIN対応UART（UART6）には，ブレーク検出モードの制御レジスタASICL6が用意されています（**図7**）．このレジスタのSBRT6ビットを‘1’にするとブレーク検出モードとなり，11ビット以上のロー・パルスを受信すると受信割り込みが発生します（**図8**）．

注1：プロトコル仕様上の規定，および技術的な原理については中編を参照．
注2：78K0/Kxシリーズでは，UARTのエラー時に専用の割り込みを発生できる．

リスト1　UARTのフレーミング・エラーを利用したブレーク検出

```
static void
LDrvUart_vosVecRxError(void)
{
    u1 u1tRxData;
    u1 u1tErrorCode;

    u1tErrorCode = DEV_u1gASIS6;

    if( (u1tErrorCode & (u1)0xFC) != (u1)0x00 )
     {
        u1tErrorCode = LCORESTAT_u1gUNKNOWNERR;
    }

    u1tRxData    = DEV_u1gRXB6;

    if( ( u1tErrorCode == LCORESTAT_u1gRXFEERR )
     && ( u1tRxData == (u1)0x00 ) )
    {
        if( LCoreCom_u1gCurMesType !=
                                 LCORECOM_u1gEVENTTRIGGERFRAME )
        {
            LCoreStat_vogSetErrorStatus( LCORESTAT_u1gRXFEERR
);
        }
        LCoreHead_vogSetSynch();
    }
    else if( (LCoreCom_u1gCurMesType ==
                                 LCORECOM_u1gEVENTTRIGGERFRAME)
          || (LCoreCom_u1gFsm == LCORECOM_u1gFSMRECID)
          || (LCoreCom_u1gFsm == LCORECOM_u1gFSMSYNCH) )
    {
        LCoreHead_vogSetBreak();
    }
    else
    {
         LCoreStat_vogSetError(u1tErrorCode);
    }

    return;
}
```

FEエラー＆受信データ0x00
→ブレーク（フレーム送受信中）

FE0，OVE0以外は
ありえないエラー

イベント・
トリガ・
フレーム
以外なら，
ブレークも
エラーとして
登録する

1バイト受信

同期バイト受信待ち処理

SYNCHの場合

RECIDの場合

イベント・トリガ衝突発生時

ASICL6

暗号	7	6	5	4	3	2	1	0
	SBRF6	SBRT6	0	1	0	1	DIR6	TXDLV6

SBRF6	SBF受信状態フラグ
0	POWER6=0およびRXE6=0に設定したとき， またはSBF受信が正常終了したとき
1	SBF受信中

SBRT6	SBF受信トリガ
0	―
1	SBF受信トリガ

DIR6	先頭ビットの指定
0	MSB
1	LSB

TXDLV6	TxD6出力反転許可/禁止
0	TxD6通常出力
1	TxD6反転出力

図7　UART6のブレーク検出モードの制御レジスタASICL6

通常は，ブレーク受信モードの開始トリガであるSBRT6ビットのみ使用する．

リスト2　ブレーク検出機能を用いた処理

SBRT設定処理のプログラム．UART6のブレーク受信補助機能を有効にする．

```
void
LDrvUart_vogSetSBRT(void)
{
     DEV_b1gSBRT6 = 1;
     return ;
}
```

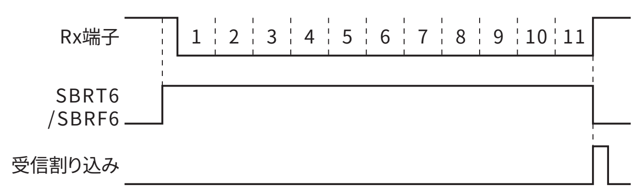


図8　ブレーク検出時の動作

SBRF6ビットを読むことで，ブレーク受信中を検知できる．

リスト3　受信ハンドラ

```
static void
LDrvUart_vosVecRx( void )
{
  u1 u1tRxData;

  u1tRxData = DEV_u1gRXB6;

  LDrvTimer_vogResetCount(ConfLin_u2gBUSTIMEOUT);
  switch(LCoreCom_u1gFsm)
  {
    case LCORECOM_u1gFSMBREAK:
      LCoreHead_vogSetSynchWithDetectBR();
      break;

    case LCORECOM_u1gFSMRECID:
      LCoreHead_vogSetResponse(u1tRxData);
      break;

    case LCORECOM_u1gFSMSYNCH:
      if( u1tRxData != (u1)0x55 )
      {
        LCoreHead_vogSetBreak();
      }
      else
      {
        LCoreHead_vogSetRecId();
      }
      break;

    case LCORECOM_u1gFSMRECDATA:
      LCoreRec_vogRecData(u1tRxData);
      break;

    case LCORECOM_u1gFSMSENDDATA:
      LCoreSend_vogCheckSendData(u1tRxData);
      break;

    case LCORECOM_u1gFSMSENDWAKEUP:
      LCoreSleep_vogWakeup();
      break;

    default:
      LCoreStat_vogSetError
                                (LCORESTAT_u1gUNKNOWNERR);
      break;
  }

  return;
}
```

1バイト受信

タイマ・カウンタ・
リセット処理（ウェイク
アップのときに再設定
するので最初に実施）

ブレーク検出

レスポンス設定処理

ブレーク
（フレーム送受信中）
後の同期確認

SYNCH中エラーは，レスポンス・エラーではない

データ受信処理

送信データ確認処理

ウェイクアップ待ち処理

そのほかはありえないエラー・セット

リスト2に，実際の設定処理を示します．該当ビットを設定するだけです．ブレークを受信すると，通常の受信割り込みが発生します．**リスト3**は，その受信ハンドラです．ブレーク受信による割り込みかどうかは，内部の状態変数で判別しています（**リスト3**の9行目）．受信後は，同期バイト処理へ移行しています（12行目）．

三つ目は，マイコンのエッジ割り込み端子を使う方法です．立ち上がり/立ち下がりの両エッジにおける割り込み，もしくはタイマのトリガを設定して，タイマのカウント値からパルス幅を得てブレークの検出を行います（**図9**）．この方法は，*RC* 発振器を用いた精度の低いクロック動作の場合や，ブレーク検出のための特別な回路を備えていない場合に有効です．

● 同期バイトにおけるボーレートの設定

ノードが水晶発振器など精度の高いクロックで動作していて，かつ使用するボーレートが固定でよい場合は，単にUARTの1バイト受信を行って0x55であるかどうかを確認すれば，同期バイトを検出できます．*RC* 発振器などを用いているため周波数補正の必要がある場合は，パルス間隔を測定してボーレート補正を行います注3．**図10**に，その処理概要を示します．

まず，UARTを停止します．立ち下がりエッジ割り込みを有効にし，その割り込み回数によって処理を行います．立ち下がりエッジ割り込み1回目でタイマをスタートし，2回目はスキップ（無処理）します．3回目の割り込みが発生すると，タイマ値を読み取って，最初の割り込みから4ビット分の時間を得ることができます．これをもとにボーレートを計算し，設定します．4回目はスキップして，5回目でエッジ割り込みを禁止し，UARTを起動します．

リスト4は実際の処理の主要部分（エッジ割り込みハンドラ内）です．すでにUARTは停止し，エッジ割り込みは許可状態で使用します．タイマは割り込みが発生してから起動すると誤差が大きくなります．そのため，計測直前に起動しておき，計測処理内ではキャプチャ・レジスタと呼ばれるエッジ割り込み時のタイマ値を保持するレジスタを

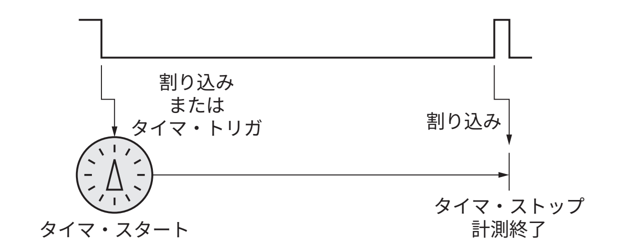


図9　パルス幅の計測
実際には，割り込みでタイマをスタートさせると，その処理分だけ誤差が生じてしまう．設計時にはタイマをあらかじめ動作させておき，割り込みの際に値を読み取って差分を計算するほうがよい．

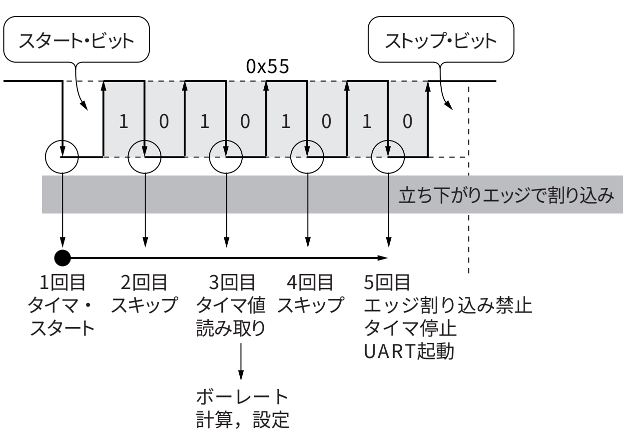


図10　ボーレート計測
処理時間に余裕を持たせるため，前半4ビット分を計測している．

注3：詳細は中編を参照のこと．

リスト4　ボーレートの算出

```
LDrvTimer_u1sEdgeCount++;

switch( LDrvTimer_u1sEdgeCount )
{
  case (u1)1:
    LDrvTimer_u2sFstCap = u2sCapNum;
    break;

  case (u1)3:
    u2sWidth = u2sCapNum - LDrvTimer_u2sFstCap;

    u2sWidth >>= 1;

    if( LDrvUart_u1gChangeBaud
                        (u2sWidth) != (u1)OK )
    {
        LCoreStat_vogSetError
                    (LCORESTAT_u1gUNKNOWNERR);
    }

    break;

    case (u1)5:
      LDrvTimer_vogDisableEdge();
      LDrvUart_vogEnable();

      LCoreHead_vogSetRecId();
      break;

     default:
      if( (u1)5 < LDrvTimer_u1sEdgeCount )
      {
         LCoreStat_vogSetError
                    (LCORESTAT_u1gUNKNOWNERR);
      }
      break;
}
```

ボーレート算出
4ビット幅計測
2ビット幅取得
エッジ割り込み禁止
UART起動
ID受信待ち処理
エッジ・カウントが5より大きいときはエラー

読み取っています．

リスト4の1行目はエッジ割り込みのカウンタ変数です．3行目のswitch文で回数に応じた処理に振り分けています．1回目（5行目）では最初のタイマ値を取得し，2回目はdefault（28行目）として無視しています．3回目（9行目）で，4ビット分の時間を取得して，ボーレート設定を呼んでいます（14行目）．

● レスポンス衝突時のブレーク受信に注意

同期バイト処理では，イベント・トリガ・フレーム時にレスポンスの衝突が起こった場合，注意が必要です．

図11は，故意にこの現象を起こしたものです．図のいちばん下は，マスタ・ノードの送信波形です．イベント・トリガ・フレームのヘッダに続いて，レスポンスを送信しています（通常，別のスレーブ・ノードがレスポンスを送信するが，ここでは実験の便宜上，マスタ・ノードに送信させている）．

中段の波形は，別のスレーブ・ノードがイベント・トリガのヘッダに対してレスポンスを送信しているものです．LINバスでは，ドミナント（“L”レベル）が優先されるので，たまたま①と②のような波形がタイミング良く重なると，LINバス上では③のように11ビットを超えるロー・パルスが生成されてしまう場合があります．ドライバ・ソフトウェアは，これをフレーム途中のブレークと解釈してしまう可能性があり，その場合，後に続くパルス列を計測してボーレート値を求めてしまいます．異常なボーレート値を算出しても±14％のガードをかけておけば，2度と通信できなくなるような状態に陥ることは免れます（下掲のコラム「ボーレート補正範囲のなぞ」を参照）．しかし，内部の処理状態は保護ID受信に遷移するので，次に来る正しいブレークをうまく認識できず，1フレーム受信しそこなってしまうかもしれません注4．

このため，ここはブレークに見えるロー・パルスについて真偽を確かめる処理を施しておくほうがよいでしょう．この処理としては，フレーム途中のブレーク検出後はボーレート補正を行わず，0x55を受信できるかどうかで直前のロー・パルスがブレークだったかどうかを調べる方法が挙げられます．この方法がもっとも負荷が小さく，かつ適切に処理できます．フレーム途中ということは，直前にボーレート補正を行っているはずなので，通信品質上は問題ありません．

再度，**リスト3**を見てください．このリストは，UARTの1バイト受信の割り込みハンドラです．9行目のswitch文で，状態（フレームの位置）に応じて処理を分岐しています．19行目が同期バイト処理のcase文であり，20行目で受信データが0x55，つまり同期バイトに相当する値かどうかを判断しています．同期バイトでなければ直前のロー・パルスはブレークではなかったと判断し，正規のブレーク待ち（状態遷移における休止状態）に遷移します．同期バイ

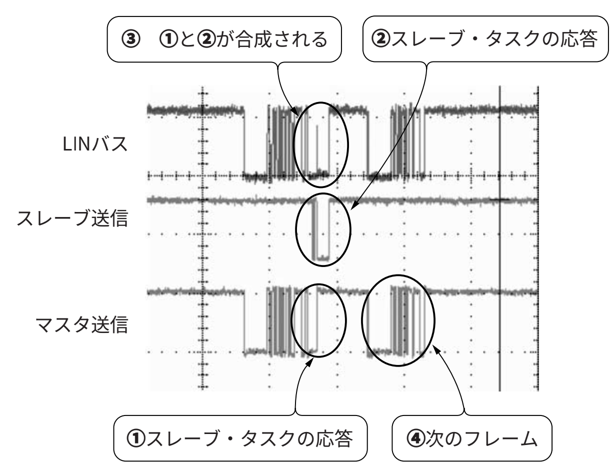


図11　衝突時のロー・パルス合成
同時に複数のノードが送信を行うと，LINバス上では負論理の論理和の波形となる．

注4：保護ID以降の受信において，ロー・パルスをブレークと判断する方法には，先に述べたようにフレーミング・エラーを利用している．このため，ボーレートが大きく（±5％以上，±14％以内）ずれると，ブレークとは認識できず単なる受信エラーとなる可能性がある．

Column 1　ボーレート補正範囲のなぞ

ボーレート補正について，LIN 1.3では±15％だった仕様が，LIN 2.0ではなぜか±14％に変更されています．ほんとうの理由はわからないのですが，マイコンによる計算のしやすさが考慮されているのかもしれません．

LIN 2.0では，オート・ボーレート（スレーブがマスタのボーレートを自動検出する機能）が考慮されています．これをサポートする場合，補正範囲の上下限も固定値ではなく，ボーレートから算出する必要があります．このとき必要な0.14倍の値は，1/8＋1/64で近似できます．つまり，シフト演算だけで補正範囲を求めることができるわけです．しかも，1/64は1/8を2回繰り返せばよいので，プログラム上では1/8，つまり右3ビット・シフト処理だけで計算できます．

トと判断された場合，次のフィールドである保護IDの処理へと遷移します．

● **保護IDは処理時間に注意**

ブレークと同期バイトを正常に処理できれば，後は基本的に普通のUART通信を行うだけです．保護IDも単に1バイト受信するだけ（のはず…）ですが，ちょっとした注意が必要です．

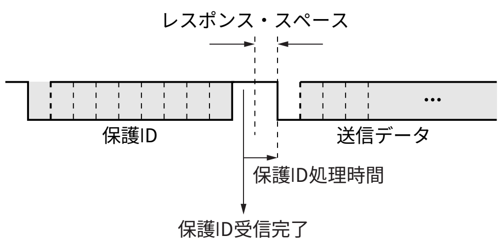


図12　保護ID処理
送信時，この処理に手間取ると，レスポンス・スペースが大きくなってしまう．

保護IDには，パリティ情報が付加されているため，これをチェックする必要があります．受信の場合，次に来るデータ・バイトをUARTが受信している間に保護IDの処理を行えばよいのですが，送信の場合はレスポンス・スペースの大きさに影響を与えてしまいます（**図12**）．

おおまかに言えば，保護ID処理から0.5ビット時間を引いた値がレスポンス・スペースとなります．マイコンが高速で，この値が負になる場合，時間待ち処理を入れる必要があります（**図5**の「送信レスポンス・スペース」状態）．

リスト5（a）は，パリティ・チェック処理をコンパイラに依存しない範囲で，C言語で記述した例です．C言語はもともとビット演算に弱いこともあり，コード効率の問題も相まって，意外とかさばる処理となってしまいます．このような場合，一部をアセンブリ言語で実装すると効果的です．**リスト5（b）**はその例です．今回の環境では，元のCソースと比べて，コード・サイズは1/4に，処理時間は1/5になりました．

リスト5　チェックサム計算処理

```
static u1
LCoreHead_u1sCheckIdParity(u1 u1tPid)
{
  const static u2 u2sPARITABLE = (u2)0x6996;
  u1 u1tTmp;
  u1 u1tPidTemp;
  u1 u1tResult;

  u1tResult = (u1)ERROR;

  u1tPidTemp = u1tPid & (u1)0x3F;

  u1tTmp = u1tPidTemp & (u1)0x07;
  if( (u1tPidTemp & (u1)0x10) != (u1)0x00 )
  {
    u1tTmp |= (u1)0x08;
  }

  if( (u2sPARITABLE & ((u2)1 << u1tTmp)) != (u2)0)
  {
    u1tPidTemp |= (u1)0x40;
  }

  u1tTmp = ( u1tPidTemp >> (u1)3 ) & (u1)0x07;
  if( (u1tPidTemp & (u1)0x02) != 0 )
  {
    u1tTmp |= (u1)0x08;
  }

  if( (u2sPARITABLE & ((u2)1 << u1tTmp)) == (u2)0 )
  {
    u1tPidTemp |= (u1)0x80;
  }

  if( u1tPidTemp == u1tPid )
  {
    u1tResult = (u1)OK;
  }

  return u1tResult;
}
```

0110 1001 1001 0110
奇数パリティ演算テーブル
パリティを除いた値を保存
ビット0，1，2，4を集める
テーブルから演算値算出
ビット6にパリティ値を反映
ビット1，3，4，5を集める
ビット7にパリティ値を反映

（**a**）C言語による記述

```
  public    _LCoreHead_u1sCheckIdParity

  OK        equ        0H
  ERROR     equ        0FFH

@@CODE    CSEG

_LCoreHead_u1sCheckIdParity:
  push    hl
  push    ax
  xch     a, x
  mov     c, #OK

  mov1    cy, a.0
  xor1    cy, a.1
  xor1    cy, a.2
  xor1    cy, a.4
  xor1    cy, a.6
  bc      $error

  mov1    cy, a.1
  xor1    cy, a.3
  xor1    cy, a.4
  xor1    cy, a.5
  not1    cy
  xor1    cy, a.7
  bnc     $fin

error:
  mov        c, #ERROR
fin:
  pop    ax
  pop    hl
  ret

  END
```

引き数u1tPidは
xレジスタで渡される

（**b**）アセンブリ言語による記述

3．データの送受信とスリープ処理

ヘッダ処理が完了すると，保護IDの値に応じてレスポンスの送受信を行います．この部分は，動作そのものはごく普通のUARTによる通信です．注意するポイントは，メッセージ・バッファへのアクセス競合です．

● データ送受信時はアクセス競合に注意

図13は，スレーブから送信する場合のメッセージ・バッファのようすを単純化したものです．API（application programming interface）からは時間順にメッセージ・バッファにデータを書き込んでいきます．一方，LINバスからもこれとは非同期に送信要求が来ます．そのため，あるタイミングでドライバからの読み出しが発生します．設計上なにも考慮していないと，新旧混在したデータを送信してしまうことになります．逆に，スレーブが受信の場合も同じような問題が起こります．データが新旧混在しても問題ないようなシステムでないかぎり，なんらかの対策を行う必要があります．

図14は，その対策例です．メッセージ・バッファは仕様上最大16個持てることになっていますが，それに加えて一時バッファ（図中，バッファ番号がTとなっている箇所）を設置します．APIからメッセージ・バッファにデータを書き込む場合，最初にいったん一時バッファに書き込みます．このとき，LINドライバ側から，APIが書き込んでい

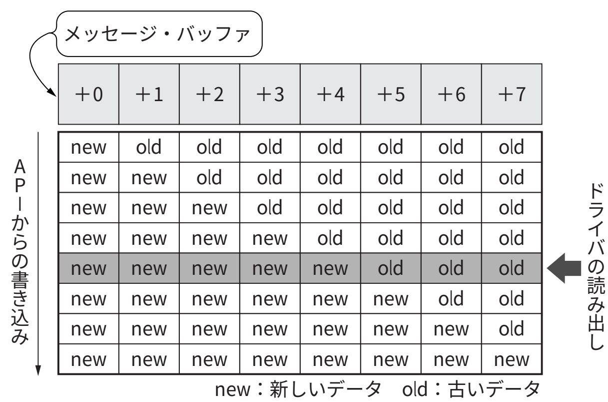


図13　バッファ・アクセスの競合
APIからの書き込み途中でLINバスからの要求が発生すると，新旧混在のデータを送信してしまう．

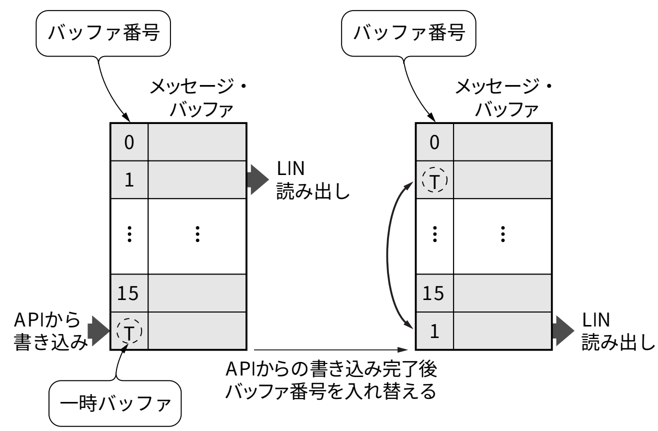


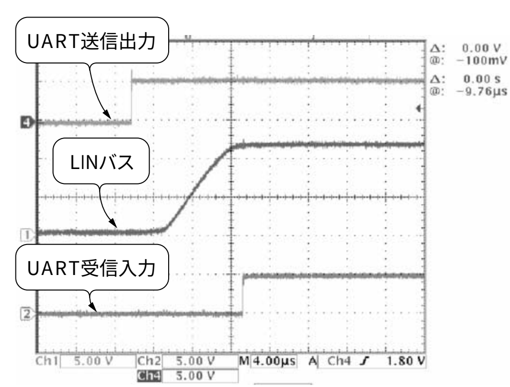
図14　バッファ制御の例
一時バッファを追加することで競合を回避する．バッファ番号の入れ替え時は，割り込み禁止などの処理が必要．

Column 2　LINの通信波形

図A-1に，LINの通信波形を示します．波形は，上からTxD端子（UARTの送信出力），LINのバス，RxD端子（UARTの受信入力）です．

LINバスの波形は，TxDからの出力をLIN仕様準拠のライン・ドライバ（回線駆動回路）を通したものです．オシロスコープの1目盛りが4μsなので，かなりゆっくりした立ち上がり波形です．車載用途なので，ノイズの発生を抑えるなどの目的で，このような仕様となっているのでしょう．

LIN対応の装置を開発する場合，とくにソフトウェア技術者は物理回線の仕様まで見ないことが多いと思います．このため，実際動作させて検査するときになって，このような波形に少なからず驚くことになります．



図A-1　LINの通信波形

リスト6　スリープ遷移

```
 1: void
 2: LCoreSleep_vogSleep(void)
 3: {
 4:     LCoreSleep_u1sWakeupRetryCnt = (u1)0;
 5:
 6:     LDrvUart_vogDisable();
 7:     LDrvTimer_vogStopTimeout();
 8:     LDrvExt_vogEnableEdge();
 9:
10:     LCoreStat_vogSetSleepFlag();
11:     LCoreCom_u1gFsm = LCORECOM_u1gFSMSLEEP;
12:
13:     ConfLin_vogCallSleepIfc(LIN_CHANEL0);
14:
15:     return;
16: }
```

ウェイクアップ・リトライ・カウント・クリア（4行目）
UART停止（6行目）
タイマ停止（7行目）
外部割り込み受け付け許可（8行目）
GoToSleepフラグを立てる（10行目）
スリープ状態に遷移（11行目）
スリープ・コールアウト（13行目）

るのと同じ番号のメッセージをアクセスにきても，**図13**のようなデータ混在問題は発生しません．APIからの書き込みが完了した時点で，メッセージ・バッファの番号を該当するものと入れ替えます．番号を入れ替える間は，割り込み禁止にするなど，LINドライバ側からのアクセスを禁止しておく必要があります．

● 実際にマイコンをスリープさせるのはアプリで

マスタからスリープ要求を受信した場合，あるいは4秒間バスが動作しなかった場合，スリープ状態に遷移します．このとき，LINドライバ・ソフトウェアは単に状態を遷移するだけです．実際にマイコンをハードウェア的にスリープ状態にするのは，アプリケーション・ソフトウェアで行います．

リスト6はスリープ遷移の処理例です．マイコンのスリープ状態は割り込みで解除されてしまうので，不要な割り込み要因となるハードウェアは停止しておきます．ただし，LINバス上のウェイクアップ信号で目覚める必要があるため，関連するリソース（ここでは外部エッジ割り込み）は許可しておきます（6行目～8行目）．

アプリケーション・ソフトウェアは，LINがスリープ状態になったことをl_ifc_read_statusというAPIで知ることができます．10行目はそのための情報を設定する手続きです．API経由でスリープ検出を行うためには，アプリケーション・ソフトウェアは定期的にこのl_ifc_read_statusを呼び続ける（ポーリングする）必要があります．それは少し不便なので，今回作成したドライバ・ソフトウェアでは，スリープ移行時にアプリケーション・ソフトウェア内の処理モジュールを呼び出す（コールアウト）処理を追加しています（13行目）．つまり，ここで呼び出しているConfLin_vogCallSleepIfcの実体は，アプリケーション・ソフトウェア側にあります．

参考・引用*文献

(1) * LIN Consortium ; LIN Specification Package Revision 2.0, Sep. 23, 2003.
(2) NECエレクトロニクス；78K0/KC1+ユーザーズマニュアル第2版（U16961JJ2V0UD00）.

たち・のぶゆき
NECマイクロシステム（株）

<筆者プロフィール>
舘 伸幸．組み込み系ソフトウェア・エンジニア．1983年入社以来，デバイス会社でソフトウェアひとすじ．